Hierarchical Autoscaling of Kubernetes Pod's

(RADIO 2023)

Abhijit Das, Rahav Vembuli, Dimple Raja Vamsi Kosaraju, Rida Zuber, Tanya Aggarwal
{abhijitd2, rvembuli, kosarajud, ridaz, aggarwalta}@vmware.com

***Abstract*—Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services. It has become the de facto standard to orchestrate, deploy, and operate containerized applications. Applications are adopting microservices architecture, which enables scaling of the application as per the workload requirements. By autoscaling microservices we achieve optimal use of resources for handling varying loads. Microservices-based architecture hosted on Kubernetes has Pod as the smallest execution unit, and the autoscaling mechanisms provided by Kubernetes operate on this unit to scale an application.**

In this paper, we propose "Hierarchical Autoscaling for Kubernetes Pods" an autoscaling mechanism by which the application microservices are scaled as per their weighted relationship with other microservices in the application pipeline hierarchy. Hence, rather than autoscaling the microservices reactively, this paper will explore a mechanism on how microservices constituting an application are autoscaled as per hierarchical dependency between them.

Keywords: Kubernetes, Autoscaling, Low-latency applications.

I. Introduction

Kubernetes provides two different mechanisms to autoscale. an application. These mechanisms work on the smallest unit in a Kubernetes cluster, i.e. Pods. Below are the two autoscaling mechanisms.

- **Horizontal Pod Autoscaler (HPA)** - scales the number of replicas of an application.
- **Vertical Pod Autoscaler (VPA)** - scales the resource requests and limits of a container.

We are not considering cluster autoscaling for this study. as it involves scaling in or out the underlying infrastructure hosting the Pods. This paper will consider only the autoscaling of Kubernetes pods.

Kubernetes' autoscaling mechanisms, HPA and VPA, use a decentralised threshold-based policy that requires setting thresholds on system-oriented metrics (i.e., CPU utilization, Memory). On breaching the threshold values, the autoscaling state machine is triggered, resulting in expanding the resource for a Pod or creating new replicas of the Pod. These autoscaling mechanism operates on each Pod in isolation. While these autoscaling mechanisms perform well for autoscaling microservices individually, with the varying workload requirements for an application, they are a reactive method of scaling and thus slower in scaling in/out the complete data pipeline of an application. Hence, if an application receives a burst of requests, the current autoscaling mechanisms would be slower to react by scaling out the microservices. Thus, the application will not be able to serve the complete workload burst and will be dropping most of the workload requests.

A. *Latency sensitive Applications*

A latency-sensitive application is an application that needs to react "fast" to specific events. Latency is defined as the time between the occurrence of an event and its handling. Example: A media/streaming player needs to react "fast" to incoming media packets (network/file) to transform them and bring them somehow to the audio output. If the application has high latency in this process, it will have drop-outs in the media player.

With the advent of 5G, augmented reality (AR) and virtual reality (VR) applications can provide both immersive and more interactive experiences due to the low latency of 5G networks. In industrial applications, for example, a technician wearing 5G AR goggles could see an overlay of a machine that would identify parts, provide repair instructions, or show parts that are not safe to touch. The opportunities for highly responsive industrial applications that support complex tasks will be extensive.

These low-latency applications are bound to be hosted on Kubernetes clusters, either on the edge or on the public cloud. As mentioned earlier, the current Kubernetes autoscaling mechanism is not suitable for serving latency-sensitive apps as they are decentralised and use a reactive method to scale the application. Until the complete application pipeline is scaled out to handle the workload burst, the workload request will fail at different stages of servicing in the application pipeline. Latency-sensitive applications cannot be scaled seamlessly in terms of response time when the application is scaled based on the system-oriented metrics of a Pod. Moreover, being a fully decentralised solution, it will lead to frequent and uncoordinated application re-configurations.

II. Hierarchical Pod Autoscaling

Most of the typical cloud native applications are deployed as interdependent microservices and the application's microservices are interconnected via the data pipeline . The data pipeline between the microservices also reflect the dependency tree between the application microservices. The current kubernetes autoscaling mechanisms do not provide an approach to coordinate the scaling operations of an application made up

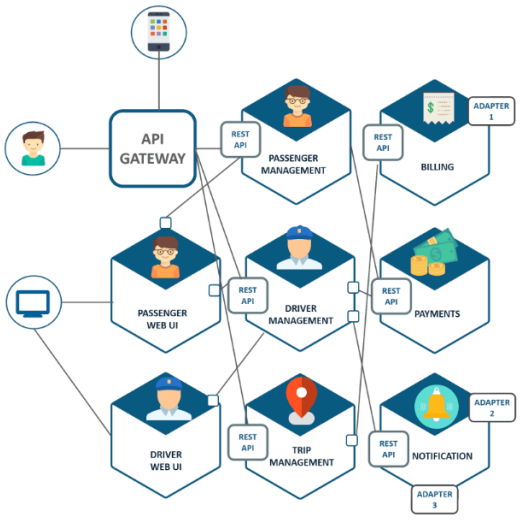

Fig. 1: Cab booking Application Pipeline.

of interdependent microservices, as HPA and VPA autoscaling mechanism operate on the Pod level.

Figure.1 depicts a typical application with a hierarchical microservice architecture.. To better illustrate the need for a hierarchical scaling solution, lets take a look at a Cab booking service. The cab booking service hosts a API gateway service, which receives requests from clients. The API gateway service based on business logic forwards the request down the multitude of dependent services. Here API gateway service acts as the root node for the hierarchy tree and the leaf nodes are formed by the notification, billing and payment microservices. In case of workload variations, the root node i.e the API gateway would be the first microservice to be receive the workload bursts. If any of the current autoscaling mechanisms are deployed then the API gateway would scale out to handle the burst , but the microservices residing in leaf nodes will not scale out until the threshold limits for that particular Pod is breached. The microservices residing in leaf nodes will scale out the Pods based on the requests that the Pod has to serve and the resources available to the Pod. As a result when the Pod eventually runs out of resources, autoscaler would kick in and try to scale out the Pod. This now adds a significant delay in scaling out the complete pipeline for processing the additional workload. Each microservice of the application independently scales in/out thus making scaling of the application disparate and unbalanced. The microservices that make the ”CAB booking application” shown in Figure.1 can be represented by a 3 level hierarchy tree as seen in Figure 2.

Today’s autoscaler solutions such as HPA and VPA are reactive in approach, meaning they trigger scale in/out operation only when the Pod is in need for more resources. This adds a significant latency to workloads where the requests either need to be queued or lost while the autoscaler scales the Pods. This is clearly undesirable as for enterprise or low latency applications we need to ensure availability of the application to the highest degree. For instance, at peak hours a rider might have to wait a significant time while trying to book a cab if the billing microservice is unable to handle the load and as a result has to be scaled out. While the billing service is scaling out, the rider will have to join a queue of riders waiting for a cab potentially increasing the time taken to book a cab.

A. *Weighted Hierarchy for application microservices*

One of the critical pieces of hierarchical autoscaler solution is to determine the extent to which each leaf node needs to be scaled for a given root node. We propose to assign weights to each leaf node in relation to its parent node. Application owner can assign weights to microservice in each leaf node a value that determines how much the microservice in the leaf node should scale in/out with respect to its parent node. This relationship will traverse from the last of the leaf nodes to the root node of the hierarchy. The weight assigned to a leaf node will be represented by a floating point number. For instance, in the example of cab booking application, the application owner can assign the following weights.

- API Gateway (rootnode) - no weights are assigned
- Passenger Web UI (leaf node, Level 2) - 2
- Trip Management (leaf node, Level 3) - 1

This would mean there could be 2 times as many passenger web UI Pods as the API gateway Pod. In the same line, there would be the same number of Trip management pods as the Passenger web UI Pods. Application owners can determine a reasonable weight to be assigned to each leaf node based on the empirical data available with the application owner. One of the reasonable method to determine the weight of a leaf node is to run the application through simulations that could depict real time scenario. An extension to this could be where a ML model could be employed to determine the desired weight for each leaf node with respect to its parent node.

III. End-to End Solution

”Hierarchical Pod Autoscaling” will be using ”Horizontal Pod Autoscaling ” as the base mechanism for scaling in/out the application pipeline. But instead of having distributed HPA policies defined for each deployment of the application microservices, we will be defining only a single ”HPA” policy on the root node of the application. The root node is the microservice which would be the first one to detect the workload variations for e.g ”API Gateway ” for the Cab booking application discussed above. For more complex application pipelines multiple hierarchies trees can be identified which do

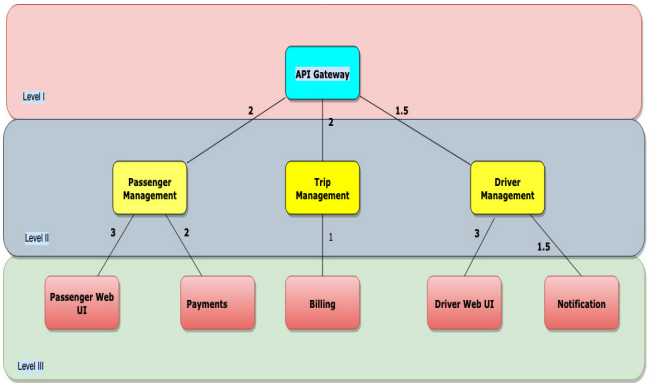

Fig. 2: Cab Booking Application Microservices Weighted Hierarchy.

not have any circular and overlapping dependencies, and for each, a different hierarchical autoscaling policy can be defined.

As described in the previous section we would need a mechanism to capture the weighted hierarchy between the microservices. We propose a new "custom resource" to capture this relationship between the microservices of an application. Below is the proposed "Hierarchical Autoscaling " CR to capture the weighted hierarchical tree.

```
type ApplicationName            string
type ApplicationWeightFactor    float32

type HeirarchicalAutoscaler struct {
        metav1.TypeMeta
        // Standard object metadata.
        metav1.ObjectMeta
        // Specification of the behavior of the heirarchical autoscaler.
        Spec    HeirarchicalAutoscalerSpec
        // Current information about the heirarchical autoscaler.
        // +optional
        Status HeirarchicalAutoscalerStatus
}

type HeirarchicalAutoscalerSpec struct {
        // RootHorizontalAutoscalerSpec should be the HPA name.
        RootHorizontalAutoscalerSpec string
        // Name of the Root micro service Application.
        RootName                    ApplicationName
        // Details of the micro service Applications.
        Nodes                       []ApplicationSpec
        // Dependency Graph of the whole application.
        DependencyGraph     map[ApplicationName]ApplicationDependencies
}

type ApplicationSpec struct {
        // Name of the micro service Application.
        Name     ApplicationName
        // A label query that determines the set of pods controlled by
        // the heirarchical autoscaler.
        Selector *metav1.LabelSelector
}

type ApplicationDependencies struct {
        // Details of all the dependencies in the mirco rservices.
        Dependencies []ApplicationWeightFactor
}

type ApplicationWeightFactor struct {
        // Name of the micro service Application.
        Name    ApplicationName
        // Wight factor with the parent mirco service application
        weight ApplicationWeightFactor
}

type HeirarchicalAutoscalerStatus struct {
        // The time when the status was last refreshed.
        LastUpdateTime metav1.Time
        // A free-form human readable message describing the status of
        // the heirarchical autoscaler.
        StatusMessage  string
}
```

Below is the sample Hierarchical Autoscaler object for the Cab booking application with all the weights.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: api-gateway-hpa
spec:
 scaleTargetRef:
   apiVersion: apps/v1
   kind: Deployment
   name: api-gateway-deployment
 minReplicas: 1
 maxReplicas: 10
 targetCPUUtilizationPercentage: 50
---
apiVersion: run.tanzu.vmware.com/v1alpha1
kind: HeirarchicalAutoscaler
metadata:
  name: sample-heirarchical-autoscaler-cr
spec:
  rootHorizontalAutoscalerSpec: api-gateway-hpa
  rootName: api-gateway
  nodes:
    - name: passenger-management
      selector:
        app: passenger-management
        env: stg
    - name: trip-management
      selector:
        app: trip-management
        env: stg
    - name: driver-management
      selector:
        app: driver-management
        env: stg
    - name: passenger-web-ui
      selector:
        app: passenger-web-ui
        env: stg
    - name: payments
      selector:
        app: payments
        env: stg
    - name: billing
      selector:
        app: billing
        env: stg
    - name: driver-web-ui
      selector:
        app: driver-web-ui
        env: stg
    - name: notifications
      selector:
        app: notifications
        env: stg
  dependencyGraph:
    api-gateway:
      - name: passenger-management
        weight: 2
      - name: trip-management
        weight: 2
      - name: driver-management
        weight: 1.5
    passenger-management:
      - name: passenger-web-ui
        weight: 3
      - name: payments
        weight: 2
    trip-management:
      - name: billing
        weight: 1
    driver-management:
      - name: driver-web-ui
        weight: 3
      - name: notifications
        weight: 1.5
```

Figure 3. depicts the workflow for "Hierarchical Autoscaling" , in which we use HPA policy for the root node to trigger the scaling in/out operation for the complete microservices dependency tree.

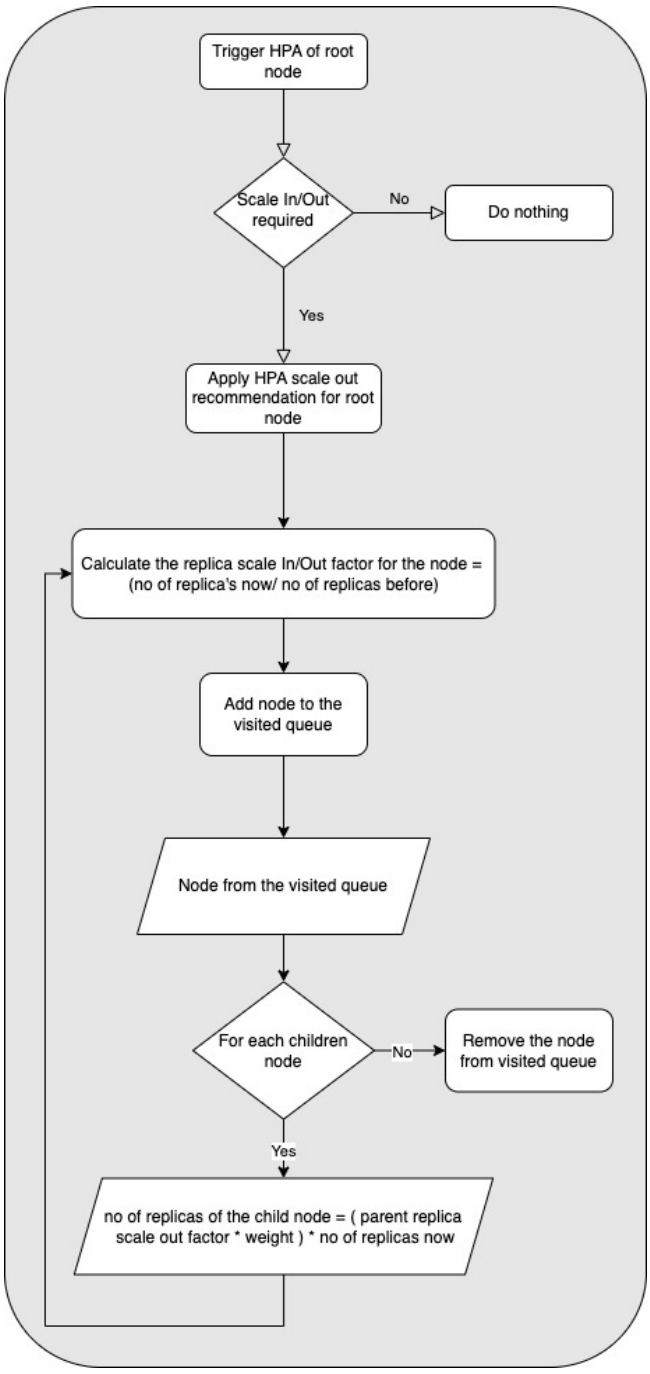

Fig. 3: Hierarchical Autoscaler Workflow.

IV. Future work

- Deriving the weighted values between the microservices of an application is needed for creating the "hierarchical autoscaling" policy. These values can be derived by conducting pre-production tests where the application pipeline is subjected to varying workload requests and measuring it against the number of failed workload requests handled. The weighted values between the microservices can be further tuned to achieve the lowest failure rates when handling varying workload bursts and This iterative process of fine-tuning the weighted values can be automated by using machine learning to derive the optimum weighted values to define the dependency between the microservices.
- As further enhancement, we would need "hierarchical autoscaling" dry run mode , by which the calculated scale in/out changes needed for a given dependency tree can be seen and edited before it can be applied on to the application pipeline by the admin.

V. Challenges

- Raising an Kubernetes Enhancement Proposal (KEP) for "Heirarchical Autoscaling for Pods" and up-streaming this new mechanism as part of sig-autoscaling project.

VI. Conclusion

For autoscaling low-latency application pipelines, the current distributed autoscaling mechanisms is not ideal because it reactive mechanism of autoscaling the application pipeline and Hence, it is slower to scale out the compleete application pipeline. Hence kubernetes needs an autoscaling mechanism that reacts faster to workload variations and failures are avoided. This can be achieved by "hierarchical Autoscaling," by which it will ensure that the entire application pipeline is scaled out/in in the shortest amount of time and thus reducing failures while handling varying workloads. Also during scaling in the application pipeline by using "hierarchical Autoscaling" compute and memory resources is optimal freed up at a faster pace.

VII. References

- F. Rossi, V. Cardellini and F. L. Presti, "Hierarchical Scaling of Microservices in Kubernetes," 2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), 2020, pp. 28-37, doi: 10.1109/ACSOS49614.2020.00023.
- Horizontal Pod Autoscaling - Kubernetes Documentation